Karlsruher Institut für Technologie

Betriebssysteme
(Operating Systems)

Prof. Dr. Michael Beigl
Jonas Kiemel, M.Sc.
Mathias Gottschlag, M.Sc.

Nachname/ *Last name*	Vorname/ *First name*	Matrikelnr./ *Matriculation no*

Scheinklausur
03.04.2019

- Bitte tragen Sie zuerst auf dem Deckblatt Ihren Namen, Ihren Vornamen und Ihre Matrikelnummer ein. Tragen Sie dann auf den anderen Blättern (auch auf Konzeptblättern) Ihre Matrikelnummer ein.

  Please fill in your last name, your first name, and your matriculation number on this page and fill in your matriculation number on all other pages (including draft pages).

- Die Prüfung besteht aus 25 Blättern: 1 Deckblatt, 19 Aufgabenblättern mit insgesamt 3 Aufgaben und 5 Blättern Man-Pages.

  The examination consists of 25 pages: 1 cover sheet, 19 sheets containing 3 assignments, and 5 sheets for man pages.

- Es sind keinerlei Hilfsmittel erlaubt!

  No additional material is allowed.

- Die Prüfung gilt als nicht bestanden, wenn Sie versuchen, aktiv oder passiv zu betrügen.

  You fail the examination if you try to cheat actively or passively.

- Sie können auch die Rückseite der Aufgabenblätter für Ihre Antworten verwenden. Wenn Sie zusätzliches Konzeptpapier benötigen, verständigen Sie bitte die Klausuraufsicht.

  You can use the back side of the task sheets for your answers. If you need additional draft paper, please notify one of the supervisors.

- Bitte machen Sie eindeutig klar, was Ihre endgültige Lösung zu den jeweiligen Teilaufgaben ist. Teilaufgaben mit widersprüchlichen Lösungen werden mit 0 Punkten bewertet.

  Make sure to clearly mark your final solution to each question. Questions with multiple, contradicting answers are void (0 points).

- Programmieraufgaben sind gemäß der Vorlesung in C zu lösen.

  Programming assignments have to be solved in C.

Die folgende Tabelle wird von uns ausgefüllt!
The following table is completed by us!

Aufgabe	1	2	3	Total
Max. Punkte	20	20	20	60
Erreichte Punkte				
Note				

Aufgabe 1: C Grundlagen

Assignment 1: C Basics

a) Betrachten Sie die folgende Funktion `calculate()`.

Consider the following function `calculate()`.

```
1 double calculate(int *ptr, int n) {
2   assert(n != 0);
3   int result = 0;
4   int *ptr2 = ptr + n;
5   while (ptr < ptr2)
6     result += *ptr++;
7   return result / n;
8 }
```

Was berechnet die Funktion? **1 pt**

What does the function calculate?

Welches Problem kann bei der Addition += in Zeile 6 auftreten? Wie könnte dieses Problem zur Laufzeit erkannt werden? **2 pt**

Which issue might occur when the addition += in line 6 is executed? How could this issue be detected at runtime?

Welches Problem kann bei der Division in Zeile 7 auftreten? Wie kann das Problem vermieden werden? **2 pt**

Which issue might occur when executing the division in line 7? How could this issue be avoided?

b) Nennen Sie eine C-Bibliotheksfunktion, die nur manchmal einen Systemaufruf durchführt. Erklären Sie, in welchen Fällen ein Systemaufruf durchgeführt wird. **1 pt**

Name a C library function which does not always perform a system call. Explain the cases in which the function performs a system call.

c) Betrachten Sie die folgende Funktionen `a()` und `b()`.

Consider the following functions `a()` und `b()`.

```
void set_configuration(struct config *);

void a() {
  struct config c;
  memset(&c, 0, sizeof(struct config));
  c.verbose = 1;
  set_configuration(&c);
}

void b() {
  struct config *c = malloc(sizeof(struct config));
  memset(c, 0, sizeof(struct config));
  c->verbose = 1;
  set_configuration(c);
}
```

Begründen Sie, warum es nicht möglich ist, eine Funktion `set_configuration()` zu schreiben, die von beiden Funktionen `a()` und `b()` korrekt aufgerufen werden kann. **2 pt**

Explain why it is not possible to write a function `set_configuration()` that both `a()` and `b()` can call correctly.

Welches Problem könnte in Funktion `a()` auftreten, wenn der `memset()`-Aufruf entfernt wird? **1 pt**

Which issue might occur in function `a()` when the call to `memset()` is removed?

Vervollständigen Sie die Funktion `a_alternative()`, sodass sie funktional äquivalent zur Funktion `a()` ist. Verwenden Sie dabei keinerlei Funktionsaufrufe. **1 pt**

Complete the function `a_alternative()` in a way that makes it functionally equivalent to the function `a()`. Do not call any functions in your solution.

```
void a_alternative() {
  c.verbose = 1;
  set_configuration(&c);
}
```

d) In dieser Aufgabe sollen Sie Funktionen zur Nutzung eines Bitvektors implementieren.

- Ein Bitvektor speichert eine Menge an Flags.
- Jedes Flag wird durch genau ein Bit dargestellt. Wenn das Flag gesetzt ist, hat das Bit den Wert 1, ansonsten 0.
- Die einzelnen Flags werden mit einem Index assoziiert.
- Ihre Bitvektor-Implementierung soll eine beliebige, aber nach der Initialisierung feste Anzahl an Flags speichern können.

In this part, you will implement functions for using a bit vector.

- *A bit vector saves a set of flags.*
- *Every flag is represented by exactly one bit in the bit vector. If the flag is set, the bit is set to 1, otherwise the bit is set to 0.*
- *Every flag is associated with an index.*
- *Your bit vector implementation should save an arbitrary number of flags that is fixed after initialization.*

Wählen Sie einen geeigneten Integer-Typ für den Bitvektor. **0.5 pt**

- Der Typ sollte so gewählt werden, dass auf heute üblichen Prozessorarchitekturen effizient damit gerechnet werden kann.
- Achten Sie darauf, dass Ihre Implementierung der übrigen Teilaufgaben korrekt mit dem hier gewählten Typ funktioniert. Schreiben Sie portablen Code.

Choose an appropriate integer type for the bit vector.

- *Choose the type so that efficient operations are possible on current processor architectures.*
- *Make sure your implementation of the remaining parts of this excercise works correctly with the integer type you choose. Write portable code.*

```
typedef
.....

.....
  bv_int;
.....
```

Vervollständigen Sie die Definition von `bpe`. Die Konstante soll die Anzahl an Bits in einem Element des Bitvektors enthalten. **0.5 pt**

Complete the definition of `bpe`. The constant should contain the number of bits per element of the bit vector.

```
/* Number of bits per element */
.....
static const size_t bpe =
.....
```

Vervollständigen Sie die Funktion `div_round_up()`, die die übergebenen Parameter dividiert und das Ergebnis aufrundet. **1 pt**

Complete the function `div_round_up()`. It should divide the passed arguments and should round the result up.

$$result = \left\lceil \frac{dividend}{divisor} \right\rceil$$

```
size_t div_round_up(size_t dividend, size_t divisor) {
.....

.....

.....

.....

.....

.....

.....

.....
}
.....
```

Vervollständigen Sie die Funktion `allocate_bitvector()`, die einen Bitvektor allokiert, der die gegebene Anzahl an Flags speichern kann. **1.5 pt**

- Der Bitvektor soll so initialisiert werden, dass keine Flags gesetzt sind.
- Allokieren Sie nicht mehr Speicher als notwendig.

Complete the function `allocate_bitvector()`. It should allocate a bit vector that can save the given number of flags.

- *In the new bit vector, all flags should be unset.*
- *Do not allocate more memory than necessary.*

```
size_t div_round_up(size_t dividend, size_t divisor);

bv_int *allocate_bitvector(size_t num_flags) {
.....

.....

.....

.....

.....

.....

.....

.....
}
.....
```

Vervollständigen Sie die Funktion `set_flag()`, die das Flag an i-ter Stelle in dem Bitvektor setzt. **2.5 pt**

Complete the function `set_flag()`. It should set the bit vector's flag at position i.

```
void set_flag(bv_int *vec, size_t i) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
.....
```

Vervollständigen Sie die Funktion `find_first_set()`, die den Index des ersten gesetzten Flags im Bitvektor zurückgibt. **4 pt**

- Der Parameter `num_flags` ist der Wert, der zuvor an `allocate_bitvector()` übergeben wurde.
- Falls keine Flags gesetzt sind, soll `num_flags` zurückgegeben werden.

Complete the function `find_first_set()`. It should return the index of the first set bit in the bit vector.

- *The parameter `num_flags` is the value that was previously passed to `allocate_bitvector()`.*
- *If there are not any flags set, the function should return `num_flags`.*

```
size_t find_first_set(const bv_int *vec, size_t num_flags) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
```

```
}
```

Total:
20.0pt

Aufgabe 2: Erkennung von Dateiänderungen

Assignment 2: Detection of Modified Files

Sie sollen ein Programm schreiben, das eine Liste von Dateien betrachtet und ihr Änderungsdatum mit dem auf einem Server gespeicherten Datum vergleicht und dadurch Änderungen der Dateien erkennt. Das Programm soll die folgenden Eigenschaften haben:

- Die Liste der zu überprüfenden Dateien wird aus einer Textdatei gelesen.
- Das vorherige Änderungsdatum wird vom Server durch Kommunikation über POSIX Message Queues abgefragt.
- Binden Sie die in den Teilaufgaben notwendigen C-Header in dem gekennzeichneten Bereich ein.
- Außer in Teilaufgabe a) müssen Sie keine Fehlerbehandung implementieren. Sie können in den restlichen Teilaufgaben davon ausgehen, dass bei Systemaufrufen und Funktionen der Standardbibliothek keine Fehler auftreten.
- Geben Sie jegliche in ihrem Code angeforderte Resourcen wieder frei.

Your task is to write a program which analyzes a list of files and compares their modification date with a modification date stored on a server in order to detect changes to the files. The program shall have the following properties:

- *The list of the files to be checked is read from a text file.*
- *The previous modification date is fetched from the server via communication through POSIX message queues.*
- *Include necessary C headers in the marked area.*
- *Except in subtask a) you do not have to implement error handling. In the other subtasks, you can assume that system calls and standard library functions do not fail.*
- *Free all resources allocated in your code.*

```
/* include statements for the required C headers */
```

```c
#define MAX_PATH 256

/* global variables */
/* name of the message queue to send the path of a file to the server */
const char *mq_path_name = "/mq-path";
/* name of the message queue to receive the modification date from the server */
const char *mq_mtime_name = "/mq-mtime";
```

a) Vervollständigen Sie die Funktion `getTimeOfModification()`, welche die Zeit der letzten Änderung einer Datei zurückgibt. **3 pt**

- Die Funktion erhält als Argument den absoluten Pfad zur Datei.
- Fehler, welche von Funktionsaufrufen innerhalb von `getTimeOfModification()` verursacht werden, sollen zu einem Rückgabewert von 0 führen.

Complete the function `getTimeOfModification()` which returns the time of the last modification of a file.

- *The function is passed the absolute path to the file.*
- *Errors caused by function calls within `getTimeOfModification()` shall lead to a return value of 0.*

```c
/* time_t is defined as long int /*
/* time_t holds the number of seconds since 00:00, Jan 1 1970 UTC */

time_t getTimeOfModification(const char *path){
.....

.....

.....

.....

.....

.....

.....

.....
}
.....
```

b) Vervollständigen Sie die Funktion `checkFileStatus()`, die das dem Server bekannte Änderungsdatum einer Datei holt und prüft, ob das aktuelle lokale Änderungsdatum abweicht. **7 pt**

- Die Funktion erhält als Argument den absoluten Pfad zur Datei.
- Die Funktion schickt den absoluten Pfad der Datei über eine POSIX Message Queue mit dem Namen "/mq-path" synchron an einen Server.
- Ist die Datei dem Server bekannt, so sendet der Server das Datum der letzten Modifikation der Datei, gespeichert als `time_t`, synchron über eine Message Queue mit Namen "/mq-mtime" an Ihr Programm. Anderenfalls sendet der Server eine nur aus Nullen bestehende Nachricht.

- Stimmt das zugeschickte Modifikationsdatum mit dem lokal bestimmten Änderungsdatum überein, so soll die Funktion 1 zurückgeben, ansonsten 0.

- Der Server übernimmt die korrekte Initialisierung der Message Queues, sodass Ihr Programm die Queues öffnen kann ohne sie zuvor erzeugen zu müssen.
- Die von `checkFileStatus()` zu sendende Nachricht soll eine Priorität von 0 haben.
- Nehmen Sie an, dass die Message Queue "/mq-mtime" vor dem Anfordern der Daten leer ist. Nachrichten können ohne Unterbrechung in einem Stück eingelesen werden.
- Alle Strings sollen null-terminiert sein. Sie können annehmen, dass Pfade aus maximal MAX_PATH Zeichen bestehen.

Complete the function `checkFileStatus()` which fetches the modification date of a file as known to the server and checks whether it matches the current local date of modification.

- *The function is passed the absolute path to the file.*
- *The function synchronously sends the absolute file path to the server via a posix message queue with the name "/mq-path".*
- *If the file is known to the server, the server sends the last modification date of the file (stored as `time_t`) synchronously to your program via a message queue with the name "/mq-mtime". Otherwise, the server sends a message consisting of zeros.*
- *If the received modification date matches the locally determined date, the function shall return 1, else it shall return 0.*

- *The server undertakes the correct initialization of the message queues meaning that your program can open the queues without creating them before.*
- *The message to be sent by `checkFileStatus()` should have a priority of 0.*
- *Assume that the message queue "/mq-mtime" is empty before requesting data. All messages can be read in one piece without interruptions.*
- *All strings shall be null-terminated. You can assume that no paths will be longer than MAX_PATH characters.*

```
/* name of the message queue to send the path of a file to the server */
const char *mq_path_name = "/mq-path";
/* name of the message queue to receive the modification date from the server */
const char *mq_mtime_name = "/mq-mtime";
time_t getTimeOfModification(const char *path);

int checkFileStatus(const char *path){
.....

.....

.....

.....

.....
```

```
}
```

Warum ist dieses Verfahren nicht geeignet, wenn mehrere Programme gleichzeitig mit dem Server kommunizieren sollen? Was müsste geändert werden, um solch gleichzeitigen Zugriff auf den Server zu ermöglichen? **2 pt**

Why is this method not suitable if multiple programs shall communicate with the server concurrently? What needs to be changed in order to allow such concurrent access to the server?

c) Vervollständigen Sie die Funktion `readFileList()`, welche die Textdatei mit der Liste der zu prüfenden Dateien einliest. Für jede aufgelistete Datei prüft die Funktion, ob die Datei seit der letzten Synchronisation mit dem Server verändert wurde. **8 pt**

- Die Funktion erhält als Argument den absoluten Pfad zur einzulesenden Textdatei.
- Jede Zeile der Textdatei enthält den absoluten Pfad zu einer zu überprüfenden Datei.
- Wenn eine Datei nicht modifiziert wurde, sollen Sie den Pfad der Datei und "not modified" auf der Konsole ausgeben. Ansonsten sollen Sie den Pfad und "modified or unknown" ausgeben.
- Verwenden Sie `fgets()`, um die Textdatei zeilenweise einzulesen. Berücksichtigen Sie, dass `fgets()` den newline-Charakter `'\n'` am Ende jeder Zeile nicht abschneidet, dieser jedoch nicht Teil des Pfades ist.
- Sie dürfen annehmen, dass jede Zeile der Textdatei maximal `MAX_PATH` Zeichen enthält.
- Leere Zeilen in der Textdatei sollen ignoriert werden.

Complete the function `readFileList()` which reads the text file with the list of files to be checked. For each listed file, the function checks, whether the file was changed since the last synchronization with the server.

- *The function is passed the absolute path to the text file to be read.*
- *Each line of the text file contains an absolute path to a file to be checked.*
- *If a file was not modified, you shall print the path of the file and "not modified" to the console. Otherwise the path and "modified or unknown" shall be printed.*
- *Use `fgets()` to read the text file line by line. Note that `fgets()` does not remove the newline character `'\n'` at the end of each line. This newline character is not part of the file path.*
- *You may assume that each line of the text file contains at most `MAX_PATH` characters.*
- *Empty lines in the text file shall be ignored.*

```
int checkFileStatus(const char *path);

void readFileList(const char *path){
```

```
}
```

Total: 20.0pt

Aufgabe 3: Multilevel Feedback Queue

Assignment 3: Multilevel Feedback Queue

Ein Multilevel Feedback Queue Scheduler (MLFB) bestimmt Prioritäten dynamisch basierend darauf, ob die Tasks ihre Zeitscheiben voll ausnutzen oder nicht. Implementieren Sie einen solchen Scheduler mit den folgenden Eigenschaften:

- Der Scheduler unterscheidet zwischen drei unterschiedlichen Prioritäten. Zwischen Task gleicher Priorität wird Round-Robin-Scheduling verwendet.
- Wenn ein Task seine Zeitscheibe vollständig ausnutzt, wird seine Priorität um eins reduziert.
- Wenn ein Task seine Zeitscheibe dreimal direkt in Folge nicht vollständig ausnutzt, wird seine Priorität um eins erhöht.
- Sie können davon ausgehen, dass zu jedem Zeitpunkt mindestens ein Task lauffähig ist, d.h. mindestens eine Queue ist nicht leer.
- Alle Zeiten werden als Vielfaches der Timer-Periode angegeben.

A multilevel feedback queue scheduler (MLFB) determines priorities dynamically depending on whether tasks fully utilize their time slices or not. Implement such a scheduler with the following properties:

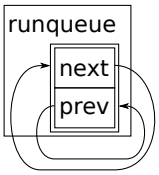
- *The scheduler distinguishes between three different priorities. Round-robin scheduling is used for tasks of equal priority.*
- *If a task fully uses its time slice, its priority is reduced by one.*
- *If a task does not fully use its time slice three times directly in a row, its priority is increased by one.*
- *You can assume that at all times at least one task is runnable, i.e., at least one queue is non-empty.*
- *All times are stated as multiples of the timer period.*

```
struct queue_entry {
    struct queue_entry *next;
    struct queue_entry *prev;
};

struct task {
    uint32_t ts;           // Remaining part of current time slice
    uint32_t prio;         // Priority of the task (index into queue array)
    struct queue_entry rq; // Scheduler queue links

    uint32_t block_cnt; // Counter of partially unused timeslices - when the
                        // counter reaches 3, increase the priority of the task.
                        // The counter is set to 0 whenever the timeslice expires.

    // ... (task context omitted for brevity)
};

// list of runnable (not blocked) tasks of the same priority
struct runqueue {
    struct queue_entry head;
};
```

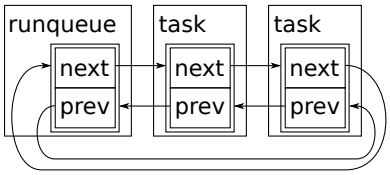
```
#define NUM_PRIO 3 // Number of different priorities
                   // (2 is the highest priority, 0 the lowest)
// Length of the time slice of each priority
const int TS_LEN[NUM_PRIO] = { 20, 10, 5 };

struct runqueue rqs[NUM_PRIO]; // MLFB queues (one per priority)
struct task *current;          // Currently executed task

void context_switch(struct task *t); // Performs a context switch
                                     // if the task has changed.
```

a) Jede der Queues ist, wie in der folgenden Abbildung zu sehen, als doppelt verkettete Liste realisiert. Leere Queues enthalten der Einfachheit halber einen Zyklus, sodass `next` und `prev` des Listenkopfes auf den Listenkopf selbst zeigen.

Each of the queues is, as shown in the following figure, implemented as a double linked list. For the sake of simplicity, empty queues contain a cycle, so that `next` and `prev` of the list head point back to the list head.



(a) Empty task queue



(b) Task queue with two tasks

Vervollständigen Sie die Funktion `from_task()`, die aus einem Pointer auf einen Task den Pointer auf den Queue-Eintrag (`rq`-Feld) des Tasks berechnet. **1.0 pt**

Complete the function `from_task()` which takes a pointer to a task and calculates the pointer to the queue entry (`rq` field) of the task.

```
struct queue_entry *from_task(struct task *t) {
.....

.....

.....

.....
}
```

Vervollständigen Sie die Funktion `from_list_entry()`, die umgekehrt aus einem Pointer auf einen Queue-Eintrag den Pointer auf den dazugehörigen Task berechnet. Nehmen Sie dabei an, dass der Compiler die Reihenfolge der Felder der Struktur nicht ändert und kein Padding zwischen den Feldern einfügt. **1.5 pt**

Complete the function `from_list_entry()` which vice versa takes a pointer to a queue entry and computes the pointer to the corresponding task. Assume that the compiler does not reorder the fields of the struct and does not insert padding between the fields.

```
struct task *from_queue_entry(struct queue_entry *e) {
.....
.....
.....
.....
}
```

Vervollständigen Sie die Funktion `append()`, die den Task `t` an das Ende der Queue `rq` anhängt. **3 pt**

Complete the function `append()` which appends the task `t` at the end of the queue `rq`.

```
struct queue_entry *from_task(struct task *t);

void append(struct runqueue *rq, struct task *t) {
.....
.....
.....
.....
.....
.....
.....
.....
.....
}
```

Vervollständigen Sie die Funktion `remove_first()`. Falls die Liste leer ist, gibt diese Funktion `NULL` zurück. Ansonsten entfernt die Funktion das erste Element aus der Liste und gibt einen Pointer auf den dazugehörigen Task zurück. **3 pt**

Complete the function `remove_first()`. If the list is empty, the function returns `NULL`. Else, the function removes the first element from the list and returns a pointer to the corresponding task.

```
struct task *from_queue_entry(struct queue_entry *e);

struct task *remove_first(struct runqueue *rq) {
.....
.....
.....
.....
.....
.....
.....
```

```
}
```

b) Vervollständigen Sie die Funktion `highest_prio_task()`, die basierend auf der Priorität den nächsten auszuführenden Task aus den Queues wählt. Der Task soll aus der Queue entfernt und zurückgegeben werden. Falls sämtliche Queues leer sind, wird NULL zurückgegeben. **3 pt**

Complete the function `highest_prio_task()` which, based on the priority, chooses the next task to be executed from the queues. The task shall be removed from the queue and be returned. If all queues are empty, NULL is returned.

```
struct runqueue rqs[NUM_PRIO]; // MLFB queues (one per priority)
struct task *remove_first(struct runqueue *rq);

struct task *highest_prio_task(void) {
    // All queues are empty, so return NULL
    // (as described in the beginning, this never happens). */
    return NULL;
}
```

c) Vervollständigen Sie die Funktion `current_task_blocked()`, die aufgerufen wird, wenn der aktuelle Task blockiert, d.h., wenn er seine Zeitscheibe nicht vollständig ausnutzt. Die Funktion soll einen anderen Task wählen und ihn in die Variable `current` schreiben. Die Funktion soll die Priorität erhöhen, wenn ein Task dreimal direkt hintereinander seine Zeitscheibe nicht vollständig ausnutzt. **3 pt**

Complete the function `current_task_blocked()` which is called when the current task blocks, i.e., wenn it does not completely use its timeslice. The function shall choose a different task and write it to the variable `current`. The function shall increase the priority if a task does not fully use its timeslice three times directly in sequence.

```
struct task *current;
struct task *highest_prio_task(void);

void current_task_blocked(void) {
    // "*current" has just blocked, replace it with a different task.
.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
    context_switch(current); // Context switch to "current"
}
```

d) Vervollständigen Sie die Funktion `timer_interrupt()`, die einmal pro Timer-Periode vom Timer Interrupt aufgerufen wird. **5.5 pt**

- Legen Sie bei Ablauf der Zeitscheibe den aktuell laufenden Task in seine Queue und wählen Sie einen anderen Task, dessen Adresse Sie in die Variable `current` schreiben. Beachten Sie dabei die Regeln für Prioritätsänderungen und ändern Sie gegebenenfalls `block_cnt`.
- Füllen Sie wo notwendig die Zeitscheibe (`ts`-Feld) der Tasks wieder auf.
- Gehen Sie davon aus, dass `highest_prio_task()` nie NULL zurück gibt.

Complete the function `timer_interrupt()` *which is called once per timer period by the timer interrupt.*

- *Upon expiration of the time slice, place the currently executing task in its queue and choose a different task, whose address shall be placed in the variable* `current`*. Follow the rules for priority changes and modify* `block_cnt` *when necessary.*
- *Replenish the tasks' timeslice (*`ts` *field) where necessary.*
- *Assume that* `highest_prio_task()` *never returns NULL.*

```
const int TS_LEN[NUM_PRIO] = { 20, 10, 5 }; // Timeslice length
struct runqueue rqs[NUM_PRIO];               // MLFB queues (one per priority)
struct task *current;                        // Currently executed task
void append(struct runqueue *rq, struct task *t);
struct task *highest_prio_task(void);

void timer_interrupt(void) {
    // Deduct one unit of time from the task's remaining time
    current->ts--;
.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
    context_switch(current); // Context switch to "current" (if changed)
}
```

Total: 20.0pt

NAME

fclose – close a stream

SYNOPSIS

#include <stdio.h>

int fclose(FILE * *fp***);**

DESCRIPTION

The **fclose**() function will flushes the stream pointed to by *fp* (writing any buffered output data using **fflush**(3)) and closes the underlying file descriptor.

RETURN VALUE

Upon successful completion 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error. In either case any further access (including another call to **fclose**()) to the stream results in undefined behavior.

ERRORS

EBADF

The file descriptor underlying *fp* is not valid.

The **fclose**() function may also fail and set *errno* for any of the errors specified for the routines **close**(2), **write**(2) or **fflush**(3).

NAME

fgetc, fgets, getc, getchar, gets, ungetc – input of characters and strings

SYNOPSIS

#include <stdio.h>

char *fgets(char * *s*, **int** *size*, **FILE *** *stream*);

DESCRIPTION

fgets() reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A **'\0'** is stored after the last character in the buffer.

RETURN VALUE

fgets() returns *s* on success, and NULL on error or when end of file occurs while no characters have been read.

NAME

fopen – stream open function

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char * *path*, **const char *** *mode*);

DESCRIPTION

The **fopen**() function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

The *mode* string can also include the letter 'b' either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with C89 and has no effect; the 'b' is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the 'b' may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-UNIX environments.)

RETURN VALUE

Upon successful completion **fopen**() returns a *FILE* pointer. Otherwise, NULL is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

The *mode* provided to **fopen**(), **fdopen**(), or **freopen**() was invalid.

The **fopen**() function may also fail and set *errno* for any of the errors specified for the routine **open**(2).

NAME

malloc, free – allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *malloc(size_t *size***);**
void free(void **ptr***);*

DESCRIPTION

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized*. If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

RETURN VALUE

The **malloc**() function return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc**() with a *size* of zero.

The **free**() function returns no value.

ERRORS

malloc() can fail with the following error:

ENOMEM
Out of memory. Possibly, the application hit the **RLIMIT_AS** or **RLIMIT_DATA** limit described in **getrlimit**(2).

NOTES

By default, Linux follows an optimistic memory allocation strategy. This means that when **malloc**() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer. For more information, see the description of */proc/sys/vm/overcommit_memory* and */proc/sys/vm/oom_adj* in **proc**(5), and the Linux kernel source file *Documentation/vm/overcommit-accounting*.

Normally, **malloc**() allocates memory from the heap, and adjusts the size of the heap as required, using **sbrk**(2). When allocating blocks of memory larger than **MMAP_THRESHOLD** bytes, the glibc **malloc**() implementation allocates the memory as a private anonymous mapping using **mmap**(2). **MMAP_THRESHOLD** is 128 kB by default, but is adjustable using **mallopt**(3). Prior to Linux 4.7 allocations performed using **mmap**(2) were unaffected by the **RLIMIT_DATA** resource limit; since Linux 4.7, this limit is also enforced for allocations performed using **mmap**(2).

To avoid corruption in multithreaded applications, mutexes are used internally to protect the memory-management data structures employed by these functions. In a multithreaded application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, glibc creates additional *memory allocation arenas* if mutex contention is detected. Each arena is a large region of memory that is internally allocated by the system (using **brk**(2) or **mmap**(2)), and managed with its own mutexes.

SUSv2 requires **malloc**() to set *errno* to **ENOMEM** upon failure. Glibc assumes that this is done (and the glibc versions of these routines do this); if you use a private malloc implementation that does not set *errno*, then certain library routines may fail without having a reason in *errno*.

Crashes in **malloc**() or **free**() are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

The **malloc**() implementation is tunable via environment variables; see **mallopt**(3) for details.

NAME

memset – fill memory with a constant byte

SYNOPSIS

#include <string.h>

void *memset(void ****s*, int** *c***, size_t** *n***);**

DESCRIPTION

The **memset**() function fills the first *n* bytes of the memory area pointed to by *s* with the constant byte *c*.

RETURN VALUE

The **memset**() function returns a pointer to the memory area *s*.

NAME

memcpy – copy memory area

SYNOPSIS

#include <string.h>

void *memcpy(void ****dest*, const void ****src***, size_t** *n***);**

DESCRIPTION

The **memcpy**() function copies *n* bytes from memory area *src* to memory area *dest*. The memory areas must not overlap. Use **memmove**(3) if the memory areas do overlap.

RETURN VALUE

The **memcpy**() function returns a pointer to *dest*.

NOTES

Failure to observe the requirement that the memory areas do not overlap has been the source of significant bugs. (POSIX and the C standards are explicit that employing **memcpy**() with overlapping areas produces undefined behavior.) Most notably, in glibc 2.13 a performance optimization of **memcpy**() on some platforms (including x86-64) included changing the order in which bytes were copied from *src* to *dest*.

NAME

strlen – calculate the length of a string

SYNOPSIS

#include <string.h>

size_t strlen(const char ****s*);**

DESCRIPTION

The **strlen**() function calculates the length of the string pointed to by *s*, excluding the terminating null byte ('\0').

RETURN VALUE

The **strlen**() function returns the number of characters in the string pointed to by *s*.

NAME

mq_close − close a message queue descriptor

SYNOPSIS

#include <mqueue.h>

int mq_close(mqd_t *mqdes***);**

Link with *−lrt*.

DESCRIPTION

mq_close() closes the message queue descriptor *mqdes*.

If the calling process has attached a notification request to this message queue via *mqdes*, then this request is removed, and another process can now attach a notification request.

RETURN VALUE

On success **mq_close**() returns 0; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EBADF
The descriptor specified in *mqdes* is invalid.

NOTES

All open message queues are automatically closed on process termination, or upon **execve**(2).

NAME

mq_open − open a message queue

SYNOPSIS

#include <fcntl.h> /* For O_* constants */
#include <mqueue.h>

mqd_t mq_open(const char ***name, int** *oflag***);**

DESCRIPTION

mq_open() creates a new POSIX message queue or opens an existing queue. The queue is identified by *name*.

The *oflag* argument specifies flags that control the operation of the call. (Definitions of the flags values can be obtained by including <*fcntl.h*>.) Exactly one of the following must be specified in *oflag*:

O_RDONLY
Open the queue to receive messages only.

O_WRONLY
Open the queue to send messages only.

O_RDWR
Open the queue to both send and receive messages.

Zero or more of the following flags can additionally be *OR*ed in *oflag*:

O_NONBLOCK
Open the queue in nonblocking mode. In circumstances where **mq_receive**(3) and **mq_send**(3) would normally block, these functions instead fail with the error **EAGAIN**.

RETURN VALUE

On success, **mq_open**() returns a message queue descriptor for use by other message queue functions. On error, **mq_open**() returns *(mqd_t) −1*, with *errno* set to indicate the error.

ERRORS

EACCES
The queue exists, but the caller does not have permission to open it in the specified mode.

ENOENT
The **O_CREAT** flag was not specified in *oflag*, and no queue with this *name* exists. *mode*.

NAME

mq_receive – receive a message from a message queue

SYNOPSIS

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio);
```

DESCRIPTION

mq_receive() removes the oldest message with the highest priority from the message queue referred to by the descriptor *mqdes*, and places it in the buffer pointed to by *msg_ptr*. The *msg_len* argument specifies the size of the buffer pointed to by *msg_ptr*; this must be greater than or equal to the *mq_msgsize* attribute of the queue (see **mq_getattr**(3)). If *msg_prio* is not NULL, then the buffer to which it points is used to return the priority associated with the received message.

If the queue is empty, then, by default, **mq_receive**() blocks until a message becomes available, or the call is interrupted by a signal handler. If the **O_NONBLOCK** flag is enabled for the message queue description, then the call instead fails immediately with the error **EAGAIN**.

RETURN VALUE

On success, **mq_receive**() returns the number of bytes in the received message; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EAGAIN
The queue was empty, and the **O_NONBLOCK** flag was set for the message queue description referred to by *mqdes*.

EBADF
The descriptor specified in *mqdes* was invalid.

EINTR
The call was interrupted by a signal handler; see **signal**(7).

EMSGSIZE
msg_len was less than the *mq_msgsize* attribute of the message queue.

NAME

mq_send – send a message to a message queue

SYNOPSIS

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned int msg_prio);
```

DESCRIPTION

mq_send() adds the message pointed to by *msg_ptr* to the message queue referred to by the descriptor *mqdes*. The *msg_len* argument specifies the length of the message pointed to by *msg_ptr*; this length must be less than or equal to the queue's *mq_msgsize* attribute. Zero-length messages are allowed.

The *msg_prio* argument is a nonnegative integer that specifies the priority of this message. Messages are placed on the queue in decreasing order of priority, with newer messages of the same priority being placed after older messages with the same priority.

If the message queue is already full (i.e., the number of messages on the queue equals the queue's *mq_maxmsg* attribute), then, by default, **mq_send**() blocks until sufficient space becomes available to allow the message to be queued, or until the call is interrupted by a signal handler. If the **O_NONBLOCK** flag is enabled for the message queue description, then the call instead fails immediately with the error **EAGAIN**.

RETURN VALUE

On success, **mq_send**() returns zero; on error, −1 is returned, with *errno* set to indicate the error.

ERRORS

EAGAIN
The queue was full, and the **O_NONBLOCK** flag was set for the message queue description referred to by *mqdes*.

EBADF
The descriptor specified in *mqdes* was invalid.

EINTR
The call was interrupted by a signal handler; see **signal**(7).

EMSGSIZE
msg_len was greater than the *mq_msgsize* attribute of the message queue.

NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

DESCRIPTION

The functions in the **printf**() family produce output according to a *format* as described below. The function **printf**() writes output to *stdout*, the standard output stream; **fprintf**() writes output to the given output *stream*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments are converted for output.

Return value

Upon successful return, these functions return the number of characters printed (not including the trailing '\0' used to end output to strings).

If an output error is encountered, a negative value is returned.

Format of the format string

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a *conversion specifier*.

The arguments must correspond properly (after type promotion) with the conversion specifier. By default, the arguments are used in the order given, where each conversion specifier asks for the next argument (and it is an error if insufficiently many arguments are given).

The conversion specifier

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

d, **i** The *int* argument is converted to signed decimal notation.

c The *int* argument is converted to an *unsigned char*, and the resulting character is written.

s The *const char* * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0').

NAME

stat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

DESCRIPTION

These functions return information about a file, in the buffer pointed to by *buf*. No permissions are required on the file itself, but—in the case of **stat**()—execute (search) permission is required on all of the directories in *pathname* that lead to the file.

stat() retrieves information about the file pointed to by *pathname*.

fstat() is identical to **stat**(), except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* file type and mode */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */

    time_t st_atime;  /* Time of last access */
    time_t st_mtime;  /* Time of last modification */
    time_t st_ctime;  /* Time of last status change */
};
```

The field *st_atime* is changed by file accesses, for example, by **execve**(2), **mknod**(2), **pipe**(2), **utime**(2) and **read**(2) (of more than zero bytes). Other routines, like **mmap**(2), may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, for example, by **mknod**(2), **truncate**(2), **utime**(2) and **write**(2) (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.